1. **Summary**

**To what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.**

The implemented testing directly aligns with the software requirements. Specifically, there were six categories of requirements listed between the two assignments.

**Class Requirements**:

* ***Contact Class***: Class member variables were created for a unique ID, first name, last name, phone number, and address for the contact object. An object instance can be instantiated with the overloaded constructor that contains the aforementioned parameters. During initialization, the constructor checks each argument against specific parameters. The unique ID, first name, last name, and phone number cannot be null or longer than 10 characters. Additionally, the address cannot be null or longer than 30 characters.
* ***Test Contact Class***: Testing the contact constructor required using a Boolean assertion (assert true). A contact object was declared and passed specific arguments that were assumed to be true. The tests corroborated this assumption. Additional tests were created to generate illegal argument exceptions when the specified argument did not meet specific requirements. Unique ID, first name, last name, and phone number were tested when they were null or longer than 10 characters. Concurrently, the address field was tested when it was null or longer than 30 characters.
* ***Task Class***: Class member variables were created for a task ID, name, and description for the task object. An object instance can be instantiated with the overloaded constructor that contains the aforementioned parameters. During initialization, the constructor checks each argument against specific parameters. The task ID and name cannot be null or longer than 10 characters. Additionally, the description cannot be null or longer than 30 characters.
* ***Test Task Class***: Testing the task constructor required using a Boolean assertion (assert true). A task object was declared and passed specific arguments that were assumed to be true. The tests corroborated this assumption. Additional tests were created to generate illegal argument exceptions when the specified argument did not meet specific requirements. Task ID and name were tested when they were null or longer than 10 characters. Concurrently, the description field was tested when it was null or long than 30 characters.
* ***Appointment Class***: Class member variables were created for a appointment ID, appointment date, and appointment description. An object instance can be instantiated with the overloaded constructor that contains the aforementioned parameters. During initialization, the constructor checks each against specific parameters. That ID cannot be null or greater than 10 characters. The date cannot be a previous date or null. And the description cannot be null or greater than 50 characters.
* ***Test Appointment Class***: Testing the appointment constructor required using a Boolean assertion (assert equals). A task object was declared and passed specific arguments that were assumed to be true. The test corroborated this assumption. Additional tests were created to generate illegal argument exceptions when the specified argument did not meet specific requirements.

**Service Requirements**:

The contact service, task service, and appointment service use in-memory data structures that support storing contacts, tasks, and appointments respectively.

* ***Contact Service***: Implements functionality to add contacts with a unique ID, delete contacts per contact ID, and update contact fields per contact ID. The updated contact method is restricted to changing the first name, last name, number, and address.
* ***Test Contact Services***: The contact service class contains three methods (add, delete, update). Testing the contact service methods required using Boolean assertions (assert true). A contact object was declared in each method and passed specific parameters. The add contact test corroborated this assumption. The delete method test corroborated the assumption that the data structure would be empty after the deletion of an object instance. The update method corroborated the assumption that only the first name, last name, phone number, and address were updated. Additionally, tests were created to generate illegal argument exceptions when the add contact method and update method violated the predefined restrictions outlined in the contact object requirements.
* ***Task Service***: Similar to the contact service, the task service implements functionality to add tasks with a unique task ID, delete task per task ID, and updates task fields per task ID. The undated task method is restricted to changing the name and description.
* ***Test Task Service***: The task service class contains three methods (add, delete, update). Testing the task service methods required using Boolean assertions (asset true). A task object was declared in each method and passed specific parameters. The add task test corroborated this assumption. The delete method test corroborated the assumption that the data structure would be empty after the deletion of an object instance. The update method corroborated the assumption that only the name and description were updated. Additionally, tests were created to generate illegal argument exceptions when the add task method and update task method violated the predefined restrictions outlined in the task object requirements.
* ***Appointment Service***: Again, the appointment service implements functionality to add appointments with a unique ID and delete appointments per appointment ID.
* ***Test Appointment Service***: The appointment service class contains two methods (add, delete). Testing the appointment service methods required using Boolean assertions (assert true). An appointment object was declared and passed specific parameters. The add appointment test corroborated this assumption. The delete method test corroborated the assumption that the data structure would be empty after the deletion of an object instance. Additionally, tests were created to generate illegal argument exceptions when the add appointment and delete appointment methods violated the predefined restrictions outlined in the appointment object requirements.

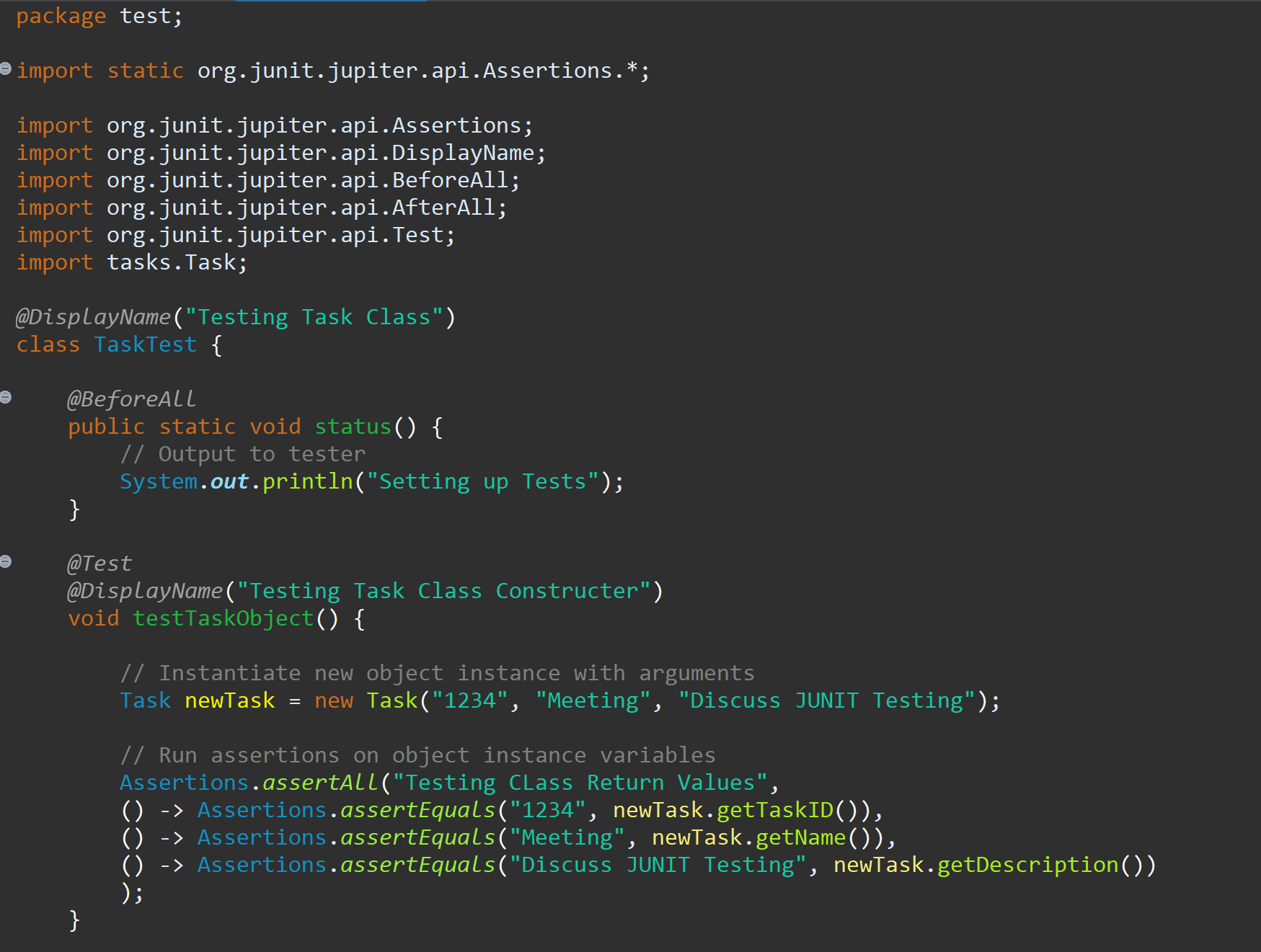
**Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**

The overall quality of my tests was written and produced at a rudimentary level. If I were to gauge the effectiveness of my test based on coverage, then I could confidently say that my tests were not effective. The recommended coverage is 80 percent. However, my test generated a 45 percent test coverage at the test level. Subsequently, the generated coverage was based on decision logic that was independent of the tested output.

Conversely, if I were to use the requirements as a measuring stick, then I could confidently say that my tests were highly effective. None of the tests failed, the expected outputs were generated, and the tests covered all of the requirements specified in the assignments. The recommended coverage is theoretical and (in some cases) impractical.

However, I am happy to report that I met or exceeded the recommended 80 percent test coverage at the class level. Contacts, Contacts Service, Task, Task Service, Appointment, and Appointment Service had a 88, 96.1, 88.2, 95.3, 87.3, and 90.7 percent coverage respectively.

***The image below will be used to answer the following prompt.***



**How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

I created a public class called Task Test in the test package. The assertions library was imported because the class was created as a new JUNIT test. Additionally, the assertions, display name, before all, after all, and test modules of the API were imported. The Task java file had to be imported before the Test Task class could have access to the object in the task package.

JUNIT testing requires the use of annotations, methods, object instantiation/initialization, and assertions. The before all annotation is an annotation that executes a method before everything. It displays output to the user that the tests are being set up. Before each method, the test annotation must be used to let JUNIT know that the method is a test. The display name annotation enhances the readability of the performed tests during analysis. It specifies what is actually being tested.

The displayed test uses a void method because it does not return anything. The method also does not have any parameters. An object is instantiated using the Task overloaded constructor and passed with arguments that are assumed to be true. The assert all method in the Assertions library is implemented with a string and a series of lambda functions. Each function performs an equality operation that compares the return value and the expected value. The method illustrates proper use of syntax, method calls, and assertion function implementations.

***The image below will be used to answer the following prompt.***



**How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**

The Task Service Test class can instantiate a Task Service object because the Test Service java file was imported. The before each annotation was used to instantiate an object with specific arguments before each test that was conducted. This eliminated redundant code within the test methods. Within the test task class add contact method, an object instance is already stored in the data structure. A for loop is used to iterate over the data structure to find a matching unique task ID.

A local variable (j) is declared and initialized with the current value of the integer being incremented. A conditional statement is used to check if the current task ID object instance is a match. When a match is found, one assert all assertion method is use to check task ID, meeting, and description object instances. This prevented multiple different tests from being created and enhanced readability.

I would like to add that the current data structure has a linear time complexity. The time complexity is not optimal. A dictionary or hash map could be used to reduce the time complexity to constant time. Additionally, a binary tree could produce a logarithmic time complexity. That being said, the code may be efficient but the data structure is the bottle neck.

1. **Reflection**

**What were the software testing techniques that you employed for each of the milestones? Describe their characteristics using specific details.**

Thomas Hamilton states that the main purpose of dynamic testing is “to test software behavior with dynamic variables or variables which are not constant and finding weak areas in software runtime environment” (Hamilton, 2023). I performed an informal review to align my testing with the assignment requirements and then conducted white box testing.

**White Box Testing**: I am familiar with internal structure of the software. The technique involves the examination of the tests based on the code written to implement a component or system. Statement testing are tests designed to force a program to execute particular statements.

I implemented a base case, forced the program to add an object instance, and verified the object instance with assertions. Also, I forced the program to execute particular decisions based on control flows (null, too long, or previous date). Then, I conducted coverage testing and achieved greater than the recommended 80 percent.

**What are the other software testing techniques that you did not use for the milestones? Describe their characteristics using specific details.**

**Static Testing**: The testing techniques used for static testing can be subcategories into two areas (reviews and static analysis). The review subtypes are informal, walkthrough technical review, and inspection. Static analysis is generally conducted using an automated tool. A dependency check tool can be used to find security vulnerabilities. The reviews can aid in defect prevention by removing ambiguities, omissions, exposures, and faults from requirements documentation.

**Black Box Testing**: Testing that occurs when the tester does not know the internal structure of the software or product (Hamilton, 2023). The technique is based on an examination of the test basis documentation (functional and non-functional). I will cover some specific details.

Equivalence Partitioning reduces the number of test cases that are written by grouping test inputs and test outputs into chunks (less than, greater than, equal to). Boundary Value Analysis is based on the principle where programmers make errors that cluster around boundaries. The technique only needs to evaluate the lower boundary values and the upper boundary values. Can be used to ensure a database field only contains specific values.

Use case testing is created from a use case diagram. Generally, the diagram is composed of an actor (outside the system) and objects (inside the system). The test attempt to replicate possible actions performed by the user and examine the outcome.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

**Static Testing**: Actions performed by the user and the associated inputs and output are validated. The team can review the database functionality requirements, hardware (computer, input/output devices), software (environment, interaction, performance), tools (IDE, programming language, API), and infrastructure (front-end, back-end, utility programs, network management).

**White Box Testing**: I used white box testing to complete the module assignments. The process is applicable to projects that require input, storage, and output. The program was forced to execute CRUD operations. The operations were tested using assertions, methods, and decision logic to control flow.

**Black Box Testing**: The company can hire a third party to conduct testing. It is better to have an independent team that does not know the underlying structure of the code to perform user acceptance testing. A team could base their tests on use case diagrams. Use equivalence portioning at the end of a sprint and boundary value analysis to verify edge cases.

**Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.**

The definition of caution is action taken to avoid mistake. Operating under this constraint, proper care was taken to ensure that my tests analyzed the specific requirements of the class and the associated service. Below is a code snippet.



From the above, we can ascertain the complexity of the code. Private member variables were needed for the arguments. I created a default constructor to instantiate an object instance and an overloaded constructer to create a custom object. I had to use the date library. However, the date library has been deprecated. The requirements stated that I had to use a deprecated library to meet the specifications.

When testing the code, I discovered that the date library will generate an illegal argument exception when the date is not entered in the proper format. This prevented the need to create another test. The interrelationship in the code was highlighted by the date storage. I had to input the date as a string. Create a new date object from the string. Then, I had to test if the parameter was null or use the new date object to test if it was a previous date. The above code demonstrates my caution (mistake mitigation), the appreciation for the code’s complexity, and the interrelationship of the tested code.

**Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.**

It is preferred that the developer does not test their own code. The process leads to bias. I cannot state that I did not have any bias. I believed the code that I wrote was correct. However, I tried to break my code as much as possible. Please refer to the code snippet below.



Bias could impact my decision to check for specific requirements. I could overlook curtain testing protocols by relying on the mistaken belief that my code is correct. Referencing the contact test, special attention was given to the naming convention. The name of an individual test was overarching. The instantiated Contacts object was passed fake information and did not contain information related to a customer or employee.

Each test was created from the requirements. The test above analyzes the object instance that was created for the correct arguments. Each test was designed to generate a specific action when the predefined specifications were not met. Every effort was made to ensure that my tests were not biased and aligned with the requirements.

**Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.**

Cutting corners while writing code can cause a loss of life (crash of a Boeing aircraft), cause a thermostat not to work in winter months, or a security breach because of a common vulnerability and exposure. Taking a short cut could cause a business to lose money, their reputation, and their reliability. The customer’s privacy and/or product experience can be compromised by inadequate due diligence on the part of the developer.

It is impossible to not incur any technical debt. An agile environment is flexible and implements client changes in the product backlog. The product owner is responsible for managing the product backlog. There are steps that the team can take to avoid technical debt. During the sprint planning, a Fibonacci number can be assigned to the use case or epic. The team decides which increment to take into the sprint.

As specific practitioner, I will allocate the time needed to finish my chunk in the allotted time. I will not accept any projects that are above my skill level. If I am assigned something beyond my skill level, then I will seek guidance from my colleagues, read documentation, and communicate my progress during daily stand ups and/or during sprint retrospectives. Additionally, I will accept projects based on how quickly I can learn and not on how much I know.

**Source**

Hamilton, Thomas. (2023). What is Dynamic Test? Types, Techniques, and Example. [Blog Post]. [What is Dynamic Testing? Types, Techniques & Example (guru99.com)](https://www.guru99.com/dynamic-testing.html)

*Software testing: An istqb-bcs certified tester foundation guide - 4th edition*. (2019). BCS Learning & Development Limited.